Function Arguments

The following are the types of arguments that we can use to call a function:

1. Default arguments
2. Keyword arguments
3. Required arguments
4. Variable-length arguments

Default Arguments

A default argument is a kind of parameter that takes as input a default value if no value is supplied for the argument when the function is called. Default arguments are demonstrated in the following instance.

Code

```
# Python code to demonstrate the use of default arguments
# defining a function
def f1( n1, n2 = 40 ):
    print("n1 is: ", n1)
    print("n2 is: ", n2)
  # Calling the function and passing only one argument
print( "Passing one argument" )
function(10)
  # Now giving two arguments to the function
print( "Passing two arguments" )
f1(10,30)
```

Output:

```
Passing one argument
n1 is:  10
n2 is:  40
Passing two arguments
n1 is:  10
n2 is:  30
```

Keyword Arguments

The arguments in a function called are connected to keyword arguments. If we provide keyword arguments while calling a function, the user uses the parameter label to identify which parameters value it is.

Since the Python interpreter will connect the keywords given to link the values with its parameters, we can omit some arguments or arrange them out of order. The function() method can also be called with keywords in the following manner:

Code

```
# Python code to demonstrate the use of keyword arguments
# Defining a function
def f1( n1, n2 ):
    print("n1 is: ", n1)
    print("n2 is: ", n2)
  # Calling function and passing arguments without using keyword
print( "Without using keyword" )
f1( 50, 30)
  # Calling function and passing arguments using keyword
print( "With using keyword" )
f1( n2 = 50, n1 = 30)
```

Output:

```
Without using keyword
n1 is:  50
n2 is:  30
With using keyword
n1 is:  30
n2 is:  50
```

Required Arguments

The arguments given to a function while calling in a pre-defined positional sequence are required arguments. The count of required arguments in the method call must be equal to the count of arguments provided while defining the function.

We must send two arguments to the function function() in the correct order, or it will return a syntax error, as seen below.

Code

```
# Python code to demonstrate the use of default arguments
# Defining a function
def f1( n1, n2 ):
    print("n1 is: ", n1)
    print("n2 is: ", n2)
  # Calling function and passing two arguments out of order, we need n1 to be 20 and n2 to be 30
print( "Passing out of order arguments" )
f1( 30, 20 )
  # Calling function and passing only one argument
print( "Passing only one argument" )
    f1( 30 )
    print( "Function needs two positional arguments" )
```

Output:

```
Passing out of order arguments
n1 is:  30
n2 is:  20
Passing only one argument
Function needs two positional arguments
```

Variable-Length Arguments

We can use special characters in Python functions to pass as many arguments as we want in a function. There are two types of characters that we can use for this purpose:

1. ***args -**These are Non-Keyword Arguments
2. ****kwargs -** These are Keyword Arguments.

Here is an example to clarify Variable length arguments

Code

```
# Python code to demonstrate the use of variable-length arguments
# Defining a function
def f1( *args_list ):
ans = []
for l in args_list:
     ans.append( l.upper() )
 return ans
# Passing args arguments
object = f1('Python', 'is', 'Good')
print( object )
# defining a function
def f1( **kargs_list ):
ans = []
for key, value in kargs_list.items():
ans.append([key, value])
return ans
# Paasing kwargs arguments
object = f1(First = "Python", Second = "is", Third = "Good")
print(object)
```

Output:

```
['PYTHON', 'IS', 'GOOD']
[['First', 'Python'], ['Second', 'is'], ['Third', 'Good']]
```

The Anonymous Functions

These types of Python functions are anonymous since we do not declare them, as we declare usual functions, using the def keyword. We can use the lambda keyword to define the short, single output, anonymous functions.

Lambda expressions can accept an unlimited number of arguments; however, they only return one value as the result of the function. They can't have numerous expressions or instructions in them. Since lambda needs an expression, an anonymous function cannot be directly called to print.

Lambda functions contain their unique local domain, meaning they can only reference variables in their argument list and the global domain name.

Although lambda expressions seem to be a one-line representation of a function, they are not like inline expressions in C and C++, which pass function stack allocations at execution for efficiency concerns.

Syntax

Lambda functions have exactly one line in their syntax:

1. **lambda** [argument1,argument2... .argumentn] : expression

Below is an illustration of how to use the lambda function:

Code

```
# Defining a function
lambda_ = lambda argument1, argument2: argument1 + argument2;
# Calling the function and passing values
print( "Value of the function is : ", lambda_( 20, 30 ) )
print( "Value of the function is : ", lambda_( 40, 50 ) )
```

Output:

```
Value of the function is :  50
Value of the function is :  90
```